REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 31 - 07 - 2004	**2. REPORT DATE** Annual Technical Progress	**3. DATES COVERED** *(From - To)* 01 July 2003-30 June 2004

4. TITLE AND SUBTITLE Language-based Security for Malicious Mobile Code	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER N00014-01-1-0968
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Fred B. Schneider, Dexter Kozen, Greg Morrisett and Andrew Myers	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cornell University Ithaca, NY 14853	**8. PERFORMING ORGANIZATION REPORT NUMBER** 39545
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Ballston Centre Tower ONE 800 North Quincy Street Arlington, VA 22217-5660	**10. SPONSOR/MONITOR'S ACRONYM(S)** ONR
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited

20041008 468

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Report summarizes progress over the past year in developing language-based technologies for defending software systems against attacks from mobile code and system extensions.

15. SUBJECT TERMS
In-lined reference monitors, proof carrying code, end-to-end security, information flow enforcement

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Fred B. Schneider
U	U	U	UU	11	**19b. TELEPONE NUMBER** *(Include area code)* 607-255-9221

Language-based Security for Malicious Mobile Code

N00014–01–1–0968

Annual Report
01 July 2003 – 30 June 2004

Fred B. Schneider (Principal Investigator)
Dexter Kozen, Greg Morrisett, and Andrew Myers (Co-Investigators)

Department of Computer Science
Cornell University
Ithaca, New York 14853

Overview

Mobile code provides a convenient, efficient, and economical way to extend the functionality and improve the performance of networked computing systems. It is an approach that has been widely embraced, as evidenced by today's operating systems, web browsers, and applications with their support for "plug-and-play", Javascript, downloaded helper applications, and executable attachments. Yet today's security architectures provide poor protection from faulty, much less from malicious, extensions. Our information systems are thus increasingly susceptible to attacks—attacks that can have devastating consequences.

This project is investigating programming language technology—program analysis and program rewriting—for defending software systems against attacks from mobile code and system extensions. The approach promises to support a wide range of flexible, fine-grained access-control and information-flow policies. Only a small trusted computing base seems to be required. And the run-time costs of enforcement should be low.

Our progress over the past year is summarized below. Details can be found in the publications whose citations are given following all the summaries. A list of DoD interactions and technology transitions appears at the end of the report.

In-lined Reference Monitors

The abstract model for security policies developed by Schneider characterized a class EM of policies meant to capture what could be effectively enforced through execution monitoring. *Execution monitors* are enforcement mechanisms that work by monitoring the computational steps of untrusted programs and intervening whenever execution is about to violate the security policy being enforced. Execution monitoring, however, can be viewed as an instance of the more general technique of *program-rewriting*, wherein the enforcement mechanism transforms untrusted programs before they are executed so as to render them incapable of violating the security policy to be enforced. Since numerous systems use program-rewriting in ways that go beyond what can be modeled as an execution monitor, a characterization of the class of policies enforceable by program-rewriters is useful.

Working with Ph.D. student Kevin Hamlen, PI's Morrisett and Schneider have developed just such a characterization. This new class of policies is called the *RW-enforceable* policies. And to date, we have connected a taxonomy of policies to the arithmetic hierarchy of computational complexity theory by observing that the statically enforceable policies are the recursively decidable properties and that the EM class of policies is the co-RE properties. We have also showed that the RW-enforceable policies are not equivalent to any class of the arithmetic hierarchy.

Execution monitors implemented as in-lined reference monitors can enforce policies that lie in the intersection of the co-RE policies with the RW-enforceable policies. The policies within this intersection are enforceable benevolently—that is, "bad" events are blocked before they occur. But co-RE policies that lie outside this intersection might not be benevolently enforceable. In addition, we have been able to show that program rewriting is an extremely powerful technique in its own right, which can be used to enforce policies beyond those enforceable by execution monitors.

Progress on Prototype IRM. Work continued on a prototype Inlined Reference Monitor (IRM) rewriter for the Microsoft's .NET and CLI intermediate language. Specifically, over the past year we developed a type system of security policies for BIL (Baby Intermediate Language), a realistic subset of CLI. A rich class of security policies could now be specified as types; the type checker ensures that a program satisfies the policy, augmenting a non-compliant program with corrective actions if necessary. Thus, the result is a compile-time way to enforce what an in-lined reference monitor can handle plus some additional policies (that are in the class of policies that require program rewriting).

Cyclone Compiler

We continued development of the Cyclone language, which is a type-safe variant of C. The goal of this work is to make it easy to port existing C code and to write new systems code to a type-safe environment. The environment guarantees the absence of attacks such as buffer overruns and format string attacks which make up the bulk of known vulnerabilities in existing systems, servers, and applications.

Work for this past year has focused on (a) improving the quality of the code generated by the compiler, (b) increasing the expressiveness of the language, and (c) increasing the assurance in the Cyclone compiler. In addition, we conducted a number of experiments to evaluate the effectiveness of the language and compiler.

With respect to code quality, we have primarily focused on static analyses for array-bounds check elimination which we earlier identified as the primary performance bottleneck. When a pointer to an array is dereferenced, we must ensure that the pointer lies within the bounds of the array. Where possible, we would like to perform this validation at compile time to avoid any run-time overhead or run-time failure. However, we discovered that many proposed approaches in the literature were in fact *unsound* due to integer overflow. We have now developed analyses that are provably sound. One analyses is based on an extension of a simple difference-constraint algorithm and another is based on linear programming.

We also integrated new features into the language to support better static checking. In particular, we added support for a limited form of dependent types which allows programmers to express relations between values (e.g., integer variable `n` holds the length of the array `A`). Dependent types let programmers capture the invariants needed to prove that certain run-time checks are unnecessary. This makes it possible to eliminate run-time type information that would otherwise be needed to support the checks. In turn, this makes interoperability with hardware and legacy code easier and less error prone.

The addition of more sophisticated types and better analyses supports better static verification of code and improved performance. However, these additions have added considerably to the size and complexity of the compiler. To mitigate the concern that bugs in the compiler could lead to a vulnerability, we have taken a number of steps ranging from good engineering practice to formal methods. For instance, we refactored the compiler to simplify its structure and introduced a suite of regression tests that exercises the critical paths in the compiler. In addition, we developed an improved model of the core type system for Cyclone and have formally proved its soundness. We are currently working to move the flow analyses and type inference out of the trusted computing base by forcing them to construct explicit proofs that can be checked by a simpler (and hence more trustworthy) checker.

Finally, we worked to evaluate the effectiveness of Cyclone in a number of settings. For instance, we ported a number of micro-benchmarks from C to Cyclone and compared the resulting performance against both C and Java. We found that on average, the Cyclone code was about 30% slower than the C code, but more than 5 times faster than the Java code. We also ported relatively large, security-critical applications, such as web servers, FTP servers, and encryption libraries to Cyclone. This has allowed us to evaluate the performance of the Cyclone code relative to the (unsafe) C code, to determine what vulnerabilities are caught by the Cyclone type-checker or run-time system, and to understand how difficult it is to port legacy applications. We found that in these large applications (which are largely I/O bound) there was almost no performance overhead and that all known (and some unknown) vulnerabilities were caught by the compiler or run-time. However, we also found that porting the code from C to Cyclone was more difficult than we expected and that further work is needed in this area.

Language-based enforcement of end-to-end security

We continued our work on analyzing and transforming programs to enforce end-to-end security properties. By identifying program dependencies (or information flows), it becomes possible to either detect insecure dependencies or automatically transform the computing system to make it secure. This has two benefits relevant to the goals of the larger project. First, we can analyze untrusted code to see whether it violates security properties. Second, we can analyze the larger software system into which this untrusted component is introduced, to understand what security guarantees are enforced even if that code misbehaves. Thus, we can recognize malicious mobile code and can also design systems that are inherently tolerant of it. Much of this work has been done in the context of Jif, an extension to the Java programming language that supports information flow analysis.

Information release and robust declassification Strong policies for information confidentiality can be enforced through static program analysis, including type systems and dataflow analysis. These analyses are able to show that no information is released from one domain to another. However, realistic programs do need to leak some information; even a program as simple as a password checker leaks some confidential information, because an attacker who tries a password learns something about the real password even when he guesses wrong. To support these programs, the Jif programming language developed by our group adds a *declassification* construct that allows explicit information release. The question then becomes what security guarantees can be offered in the presence of this powerful escape hatch.

Recently we defined a new end-to-end security property we call *robustness.* It captures the following idea: although a system may release sensitive information (intentionally), it should not be possible for an attacker to affect what information is released or whether information is released at all. In our CSFW 2004 paper, we formally characterize this property in the setting of simple programming language and give a compile-time program analysis that provably enforces the property. This program analysis turns out to be very similar to an analysis that we had already employed in our Jif/split compiler, so it also helps justify the security of our work on automatically partitioning programs for distributed systems.

Dynamic policies We have been investigating information security in systems where policies change or are computed dynamically. This capability is important for realistic computing systems. For example, when a program opens a file on the file system, it does not usually know in advance how sensitive the information in the file is; this must be discovered dynamically. Dynamic policies are also important for transmitting information through multilevel channels. However, dynamic change introduces the possibility that covert channels will be created either through inadvertent downgrading or by communicating through the choice of policy itself.

The Jif programming language has some support for dynamic policies but we have found that it is not expressive enough to build some systems of interest. Therefore, we developed a richer dynamic policy framework in the context of a simple but expressive functional programming language. We showed that the type system for this language enforces the desired security properties, preventing improper downgrading and covert policy channels. This work will be published later this summer. The key insight is to represent information security labels (representing policies) as first-class values in the language and to analyze information flow using a dependent type system in which types record what dynamic information can affect these labels. We are now planning to implement this more expressive type system as part of the Jif language.

Support for security extensions to Java To support the implementation of the various versions of Jif, we have developed an *extensible compiler framework* that makes it easy to build compilers for languages similar to Java. Reported in our paper at the 2003 Conference on Compiler Construction, this is a basic tool for supporting research in language-based security, because it makes it easy to add a broad range of new annotations or even statements and expressions. This framework has been used to construct more than fifteen variants of the Java language. Like the Jif compiler, the Polyglot framework is available for public download. It is being used for several ongoing projects outside Cornell and continues to attract interest.

Availability Information flow analysis has been widely used to characterize confidentiality and integrity properties of programs; we have been exploring how to extend it to analyze and enforce availability policies as well. Intuitively, integrity properties ensure that data will be correct if it is available, whereas availability properties ensure that data will be available but say nothing about correctness. Distinguishing between these two properties is important for obtaining an accurate security analysis, because integrity and availability behave differently in a distributed, replicated setting. For example, simple replication improves availability but harms integrity and confidentiality because there are more sites to attack. More complex replication and voting schemes introduce a rich space of tradeoffs. We have defined an availability analysis for a simple programming language and are now exploring a unified framework for analyzing confidentiality, availability, and integrity in a distributed, replicated system.

Avoiding Malicious Firmware

Boot firmware runs in privileged mode prior to the start of most security services and before the operating system has booted, and it has up to now been necessary to accept boot firmware as part of the trusted code base. Unfortunately, boot firmware often includes on-board device drivers supplied with the devices. These devices are mass-produced all over the world by third-party manufacturers, who may not even be known to the end consumer. Thus boot firmware is a plausible avenue for the widespread and covert introduction of malicious code. BootSafe guards against this by statically checking on-board drivers against a built-in security policy each time they are loaded.

The BootSafe system is based on Open Firmware, a widely used standard for boot firmware. Sun Microsystems and Apple both use boot firmware that conforms to this standard. Open Firmware-compliant systems include an interpreter or virtual machine for *fcode*, a lightly compiled form of the Forth programming language.

The BootSafe system enforces a three-tiered baked-in safety policy for device drivers consisting of (i) basic type safety, memory safety, stack safety, and control-flow safety at roughly the level provided by the Java bytecode verifier; (ii) a device encapsulation policy that prevents device drivers from operating other devices except where explicitly allowed by the policy (for instance, a PCI device may communicate with the PCI bus to which it is attached), and then only through published interfaces; and (iii) a structural safety policy, which enforces that code supplied by vendors will interact with Open Firmware services through the published interface.

To ensure type safety, our verifier relies on the fact that drivers are compiled from a high level language, namely Java. The BootSafe prototype consists of three interlinked elements: J2F, a Java VM-to-fcode compiler; a stand-alone verifier that is trusted and part of the boot kernel; a Java API for BootSafe-compliant Open Firmware drivers; and a runtime support module.

Over the past year, we finished building the prototypes of these elements. In addition, we produced working device drivers for PCI disk and PCI net devices written in Java. These drivers can be compiled to Java bytecode with an ordinary off-the-shelf Java compiler, then further compiled to fcode with J2F. The resulting fcode passes verification with our verifier.

Static Analysis with Kleene Algebra

Kleene algebra with tests (KAT) is an algebraic system for program specification and verification that combines Kleene algebra, or the algebra of regular expressions, with Boolean algebra. One can model basic programming language constructs such as conditionals and while loops, verification conditions, and partial correctness assertions. KAT has been applied successfully in substantial verification tasks involving communication protocols, source-to-source program transformation, concurrency control, compiler optimization, and dataflow analysis. The system is *PSPACE*-complete and deductively complete for partial correctness over relational and trace models.

KAT has a rich algebraic theory with many natural and useful models: language-theoretic, relational, trace-based, matrix. Because of its roots in classical algebra and equational logic, KAT provides a mathematically rigorous foundation that subsumes many previous approaches, recasting them in a more classical algebraic framework. Hoare logic and program schematology are two examples of major theories in computer science that are subsumed by KAT.

We recently demonstrated that KAT provides a general framework for the static analysis of programs and given a construction that shows how to use KAT to statically verify compliance with safety policies specified by Schneider's security automata, a popular mechanism for the specification and enforcement of a large class of security policies. A security automaton is an ordinary finite-state automaton in which certain states are designated as *error states.* A transition to a new state may occur when a critical operation of a program is executed. Any computation of a program containing a sequence of critical operations that sends the automaton to an error state violates the policy as specified by the automaton.

The automaton can be used for runtime enforcement of the security policy as well as specification. The program code is instrumented to call the automaton before all critical operations (ones that could change state of the automaton). The automaton aborts the computation if the operation would cause a transition to an error state. This is purely a runtime mechanism. However, KAT be used to verify compliance with the security policy statically, before execution. The method uses the KAT rules to propagate state information throughout the program to all critical operations. If the verification is successful, an independently checkable proof object is produced that can be used to certify that the runtime checks are unnecessary. The method is shown to be sound in the sense that any program verified in this fashion satisfies the policy. A version of the soundness theorem with a simplified verification condition holds whenever the program is known to be total. There is also a corresponding weak completeness theorem that says that if the propositional abstraction of the program fails to verify, then there is a relational interpretation in which the program is unsafe.

The method has been used to verify an example device driver, and an interactive theorem prover for KAT, written in SML, has been developed.

Publications: July 2003 – June 2004

(1) Kamal Aboul-Hosn and Dexter Kozen. KAT-ML: An Interactive Theorem Prover for Kleene Algebra with Tests. *Proc. 4th Int. Workshop on the Implementation of Logics (WIL'03)* (Almaty, Kazakhstan, Sept. 2003), 2–12.

(2) James Cheney. The Complexity of Equivariant Unification. *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, (Turku, Finland, July 2004).

(3) James Cheney. Nominal Logic Programming. Ph.D. Thesis, Cornell University (August 2004).

(4) J. Cheney and C. Urban. System Description: Alpha-Prolog, a Fresh Approach to Logic Programming Modulo alpha-Equivalence. *Proc. 17th Int. Workshop on Unification, UNIF'03*, (Valencia, Spain, June 2003), 15–19.

(5) Matthew Fluet and Daniel Wang. Implementation and Performance Evaluation of a Safe Runtime System in Cyclone. *Proceedings of the SPACE 2004 Workshop*, (Venice, Italy, January 2004).

(6) M. J. Gabbay and J. Cheney. A Proof Theory for Nominal Logic. *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, (Turku, Finland, July 2004), 139–148.

(7) Dexter Kozen. Automata on Guarded Strings and Applications. *Matématica Contemporânea 24* (2003), 117–139.

(8) Dexter Kozen. Computational Inductive Definability. *Annals of Pure and Applied Logic 126*, 1–3 (April 2004), 139–148.

(9) Dexter Kozen. Some Results in Dynamic Model Theory. *Science of Computer Programming 51*, 1–2 (May 2004), 3–22.

(10) Dexter Kozen and Jerzy Tiuryn. Substructural Logic and Partial Correctness. *Trans. Computational Logic 4*, 3 (July 2003), 355–378.

(11) Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. *IEEE Computer Security Foundations Workshop* (Pacific Grove, CA, June 2004), 172–186.

(12) Fred B. Schneider. Least Privilege and More. *IEEE Security and Privacy 1*, 3 (September/October 2003), 55–59.

(13) Fred B. Schneider Lifting reference monitors from the kernel. *Formal Aspects of Security, FASec 2002* (London, United Kingdom, December 2002), Ali E. Abdullah, Peter Ryan, and Steve Schneider (eds.). Lecture Notes in Computer Science, Volume 2629, Springer-Verlag, New York, 2003, 1–2.

DoD Interactions and Technology Transitions

- As a consultant to DARPA/IPTO, Schneider chairs the independent evaluation team for the OASIS Dem/Val prototype project. This project funds two consortia to design a battlespace information system intended to tolerate a class A Red Team attack for 12 hours. Schneider also serves on the independent evaluation team for the new DARPA/IPTO Self-Regenerative Systems program.
- Schneider serves on the NRC CSTB committee on improving cybersecurity research. This is a 2-year congressionally mandated study.
- Schneider served on the AFRL search committee for Senior Scientist in Information Assurance Technology.
- Morrisett and Schneider each briefed the Infosec Research Council's "Research Hard Problems" study; Schneider also served as a reviewer for the final resport.
- Myers, Kozen, and Schneider each participated in an advanced computer science lecture series at AFRL/Rome.
- Further public releases of Myers' Jif compiler have been made available at the Jif web site, http://www.cs.cornell.edu/jif. The Jif language extends the Java programming language with support for information flow control. The Jif compiler is implemented on top of the Polyglot extensible compiler framework for Java. The Polyglot framework has also been released publicly at http://www.cs.cornell.edu/projects/polyglot, and researchers at Princeton University are using this framework in their own research. The releases of both Jif and Polyglot are provided as Java source code and work on Unix and Windows platforms.

- AT&T research is working with us to develop the Cyclone language, compiler, and tools. The source code for the compiler and tools are freely available and may be downloaded from the web. In addition, researchers at the University of Maryland, the University of Utah, Princeton, and the University of Pennsylvania, and Cornell are all using Cyclone to develop research prototypes.
- Public releases of Kozen's KAT interactive theorem prover have been made available at the project website http://www5.cs.cornell.edu/kamal/kat/ for Mac OS X, Linux, Solaris, and Windows platforms.